Optimizando los Indices Invertidos para la Búsqueda de Palabras en Sistemas P2P

Victor Hugo Batista Juan Manuel Fuertes Javier Echaiz Jorge Ardenghi

Laboratorio de Investigación de Sistemas Distribuidos (LISiDi)
Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur, Bahía Blanca (8000), Argentina
batista.victor.hugo@gmail.com

juanm_fuertes2@hotmail.com {je,jrap}@cs.uns.edu.ar

Resumen

La búsqueda de objetos a través de palabras es una componente crítica en un sistema P2P. Este no es un tema trivial en las redes estructuradas como las DHTs, dado que solo permiten búsqueda de palabras exactas. Se han propuesto diversas soluciones a este problema, pero es difícil encontrar una eficiente. Los índices invertidos son una solución sencilla, intuitiva, donde encontrar los resultados es directo, pero en su forma más simple presentan varias falencias, como carga desbalanceada, hot spots, pobre tolerancia a fallas y no permitir la búsqueda de palabras inexactas. En este paper presentamos una solución utilizando índices invertidos, donde se contemplan los problemas planteados y se resuelven con diversas técnicas, obteniéndose un sistema robusto y eficiente, que cumple con los requisitos de un motor de búsqueda para sistemas peer-to-peer.

Palabras Clave: Peer-to-Peer, DHT, Búsqueda, Indices invertidos, Hot Spots, Palabras Inexactas.

1. Introducción

No es novedad que los sistemas Peer-to-Peer han revolucionado la computación distribuida, desde los sistemas centralizados como Napster, pasando por los descentralizados sin estructura como Gnutella hasta los sistemas estructurados como Chord [1, 2], Kademlia [3], CAN [4], etc.

La mayoría de los sistemas estructurados utilizan tablas hash distribuidas (DHT) como base. Los objetos están asociados a los nodos a través de un identificador (nodos y objetos tienen identificadores, nodoID y key respectivamente, con el mismo formato y longitud). La key es calculada a partir de un hash sobre propiedades del objeto (nombre, contenido, etiquetas, etc). Para localizar un objeto, se debe conocer el ID del nodo que lo posee. Por lo tanto, se debe conocer la propiedad utilizada.

Si bien las DHTs proveen un servicio de búsqueda muy eficiente (típicamente localizar un objetivo lleva O(log N) saltos), los objetos solo pueden ser localizados a partir de un identificador único, key, que no suele ser conocido por los usuarios. Comúnmente, estos desean buscar los

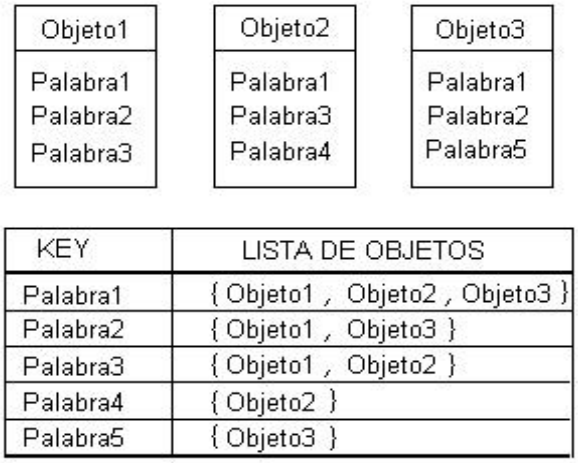

Figura 1: Un índice invertido con tres objetos y cinco palabras

objetos a partir de un conjunto de palabras claves. Para proveer búsqueda usando palabras claves, se han propuesto varias soluciones. La más común es usar un índice invertido, donde se asocia a cada palabra la lista de objetos que la contienen. La idea principal es usar el hash de cada palabra como keys, y tener un mapeo <key, lista de objetos> (una palabra puede estar contenida en varios objetos), en lugar del mapeo <key, objeto>. La búsqueda de objetos, dado un grupo de palabras p_1, p_2, ..., p_n, se realiza obteniendo la lista de objetos de los nodos cercanos a los hashes de las palabras (h(p_1), h(p_2), ..., h(p_n)) e intersectando estas listas para descartar los documentos que no contengan todas las palabras.

Este método es sencillo, intuitivo, encontrar los resultados es directo y existe una implementación (Kad) con decenas de miles de nodos conectados, que ha demostrado ser estable y efectiva a la hora de realizar consultas. A pesar de esto, los índices invertidos tienen varios inconvenientes. Entre estos, los más importantes son:

- ▷ Sobrecarga: debido a que en una consulta con k palabras se necesitan las respuestas de k nodos, donde cada respuesta contiene muchos documentos que luego serán descartados al hacer la intersección. El movimiento de tantos datos sobrecarga la red, y la operación de intersección consume demasiados recursos.

- ▷ Balance de carga: la frecuencia de las palabras (la cuenta de apariciones de palabras en objetos) varia enormemente. La distribución típicamente sigue la *ley de Zipf*, lo que significa que algunas palabras aparecen muy seguido, mientras algunas otras, aparecen raramente. Esto genera hot spots, nodos que son consultados muy frecuentemente.

- ▷ Indexing Overhead: si un objeto contiene las palabras w_1, w_2, ..., w_n, entonces crear entradas por cada palabra significa que la información acerca del objeto esta repetidamente guardada en n lugares diferentes.

- ▷ Único punto de falla: a pesar de que un objeto es indexado desde múltiples lugares, cada palabra sigue siendo manejada por un solo nodo. Cualquier falla del nodo negará todas las consultas que involucran esa palabra.

- ▷ Solo búsquedas exactas: la estructura de índices invertidos usando hash de palabras no permite buscar palabras mal escritas o con errores ortográficos.

En este paper presentamos MamUSHkA, una solución que integra y refina las propuestas de otros autores, como así también aporta ideas nuevas que aprovechan las ventajas de los índices invertidos y a su vez intenta resolver los problemas planteados. Con esto se obtiene un sistema robusto y eficiente, que cumple con la mayoría de los requisitos de un motor de búsqueda. MamUSHkA reduce el consumo de ancho de banda, complejidad de búsqueda y alivia la carga impuesta a los nodos que mantienen palabras populares; a su vez permite búsqueda de palabras especificando parámetros y soporta consultas con palabras inexactas utilizando un corrector de palabras similar al de Google (usualmente conocido como el resultado Did you mean).

2. Trabajos Relacionados

En las primeras implementaciones de búsqueda de palabras totalmente descentralizada (sistemas del estilo Gnutella), el mecanismo utilizado era realizar un broadcast con un mensaje de consulta, donde cada nodo era inspeccionado en busca del objetivo. Debido a que un broadcast descontrolado podría inundar la red con mensajes, se utiliza un TTL (Time To Live - Tiempo de Vida) para limitar la cantidad de saltos de un mensaje. Esto provoca que encontrar el objetivo no esté garantizado en el caso que el objeto se encuentre en un nodo lejano. Este mecanismo emerge del formato sin estructura de la red, donde la búsqueda se realiza a ciegas. Se han propuesto muchas mejoras para las búsquedas en sistemas sin estructura, pero ninguna muy satisfactoria.

Con las redes estructuradas con DHTs surgió un nuevo problema, estas solo soportan la búsqueda de palabras exactas. Esto se debe a que se asigna un identificador único a los objetos, obtenido de realizar un hash de sus palabras claves, para determinar su ubicación en la red. Se propusieron varios mecanismos para realizar búsquedas en DHTs, siendo los índices invertidos la estructura principal de la mayoría. Dadas las desventajas de esta estructura, se han sugerido distintas formas de afrontarlas.

En [5] se propone el uso de Bloom Filters como una forma de ahorrar ancho de banda. Cuando se realiza una consulta, en lugar de intercambiarse largas listas de documentos entre los nodos que administran los documentos asociados a cada palabra de la consulta, se intercambian bloom filters creados a partir de las listas de documentos. También se utiliza caching para guardar bloom filters recibidos y ahorrar así retransmisiones.

En [6] se utiliza el hash de múltiples palabras, relacionadas con el objeto, como una sola entrada en la tabla de índices invertidos. Cuando se busca un conjunto de palabras K, se utiliza un subconjunto k de las palabras en K para la consulta, consultando solo al nodo que mantiene la entrada para subconjunto k. Este método salva ancho de banda y acelera las búsquedas, pero a expensas de utilizar un mayor espacio de almacenamiento.

La red Kad [7] ahorra ancho de banda de forma similar al último método nombrado. En Kad cuando se desea insertar un objeto en la red, se calcula el hash de cada palabra y en los nodos encargados de estos IDs se insertan referencias al objeto junto con el resto de los metadatos del objeto (nombres, etiquetas, etc). Las consultas se realizan a partir de una sola palabra del conjunto K, es decir, se consulta un solo nodo, y este nodo se encarga de buscar, en su lista de documentos, aquellos documentos que contienen el resto de las palabras de K. Kad además utiliza republicación, que se utiliza para que la lista de documentos disponibles este actualizada. También se proponen distintas técnicas para evitar que la sobrecarga, entre estas, las más importantes son: limitar la cantidad de referencias que mantiene un nodo y utilizar un delay para la republicación.

[8] resuelve el problema de buscar palabras inexactas, calculando el vecindario de supresión

de cada palabra del documento a insertar y agregando un enlace desde los nodos encargados de mantener esas palabras. El vecindario de supresión se calcula utilizando la métrica edit-distance. De esta forma, para realizar una búsqueda, se calcula el vecindario de supresión de las palabras a buscar y se consultan los nodos encargados de ellas, para luego decidir cuales resultados son relevantes. Este método claramente impone una sobrecarga tanto en las inserciones como en las consultas.

[9] propone armar un diccionario con las palabras más frecuentes (llamado Fusion Dictionary Diccionario Fusión DF) que registre las palabras que aparecen con una frecuencia mayor a un umbral. El mecanismo utilizado es eliminar de las consultas aquellas palabras que aparecen en el DF, con lo que se pueden eliminar las largas listas de documentos de los nodos que manejan palabras muy frecuentes. De esta forma, no solo se alivia al nodo que administra una palabra frecuente, sino que también las búsquedas son más precisas, dado que en las consultas solo se utilizan palabras poco frecuentes. El método se basa en que sólo algunas palabras del lenguaje son muy frecuentes, con lo que mantener este diccionario es barato.

[10] sugiere utilizar k funciones hash diferentes para separar los peers en grupos. Usando múltiples hashes cada palabra es balanceada entre un conjunto de nodos en el sistema, con una pequeña superposición entre diferentes conjuntos de nodos, lo cual logra un buen balance de carga en términos de tráfico y de almacenamiento de claves, así como también hacer las búsquedas resistentes a las fallas.

Además de las distintas soluciones que utilizan índices invertidos, se propusieron algunas alternativas, como [11] que propone un mecanismo donde se particiona el índice por documentos entre grupos, y a su vez se particiona el índice por palabras entre los nodos dentro de un grupo. Este método busca aprovechar los beneficios de ambos mundos. En [12] se utiliza una red subyacente con forma árbol de sufijos distribuido como una capa intermedia entre la red subyacente DHT y la red subyacente semántica. Este mecanismo soporta búsqueda de secuencias de palabras.

Una alternativa muy interesante es la planteada en [13] donde arman una red lógica con forma de hipercubo por encima de la DHT. Este sistema utiliza hashes de las palabras de un objeto para setear posiciones en un arreglo de bits (que en un principio solo contiene ceros). El arreglo calculado se utiliza como el identificador del nodo del hipercubo que almacenará el documento. Para realizar búsquedas, se calcula el arreglo con los hashes de las palabras a buscar y se envía el pedido al nodo N correspondiente. Cuando N recibe el pedido, analiza si contiene documentos para esa búsqueda, sino, reenvía la consulta a los distintos nodos de su spanning binomial tree, que es un árbol que arma a partir del hipercubo. El fundamento de esto es que, si el nodo actual no tiene el documento, entonces algún nodo en el spanning binomial tree lo tendrá, dado que estos tienen más bits seteados en sus identificadores (esto es, tienen documentos con las mismas palabras buscadas más algunas otras). Si bien este sistema balancea la carga, cada objeto es referenciado por un solo nodo, no genera hots pots y permite la expansión de palabras, es ineficiente a la hora de buscar. Esto se debe a que una consulta puede requerir buscar en todo el spanning binomial tree, el cual puede ser muy grande si la cantidad de bits utilizados en el ID es de 10 o más. Además, al utilizar un solo hash por palabra del objeto y un arreglo de pocos bits para el ID, obtendrá muchos falsos positivos.

3. Modelo de Sistema Propuesto

Con el sistema de índices invertidos, gracias al mapeo <palabra, lista de objetos>, es directo obtener resultados relevantes a partir de palabras incluidas en los objetos deseados, además

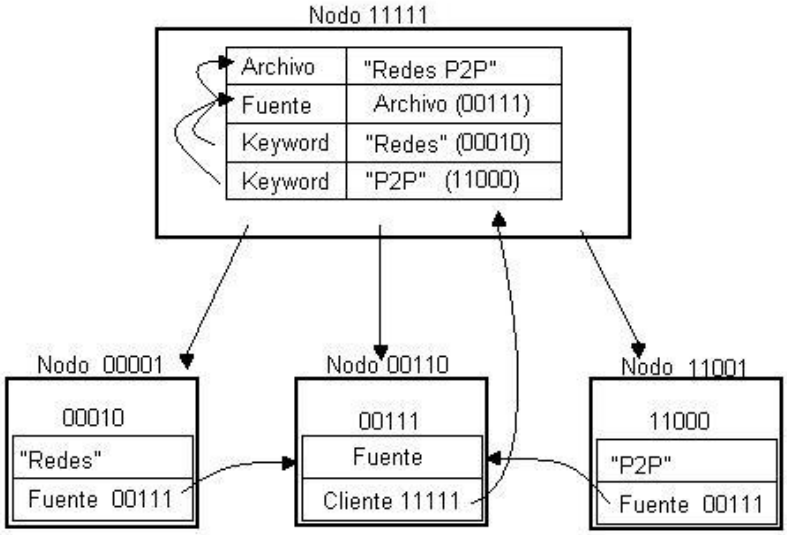

Figura 2: Presenta un ejemplo donde se desea publicar el archivo llamado "Redes P2P". El peer genera los hashes de las palabras "Redes" y "P2P", y además calcula el hash del archivo. Este nodo luego inserta metadatos en los nodos encargados de esas palabras, como así también la información de ubicación.

asegura que se encontrará el objeto, si es que este existe. Otros métodos, a diferencia de este, requieren pasos adicionales (como por ejemplo, recorrer un árbol) para obtener los objetos deseados. Además, algunos limitan el número de nodos a recorrer, lo cual puede ocasionar que el objeto no se encuentre.

Como en la red Kad, en MamUSHkA, existen metadatos e información de ubicación. Los metadatos son identificados por el hash de una palabra, contienen información del archivo (como los tags ID3 de los mp3) y un puntero al nodo con la información de ubicación del archivo. La información de ubicación, mantiene punteros a los nodos con el archivo real. El nodo que guarda la información de ubicación, se elige a partir del hash del archivo fuente. De esta forma, archivos de igual contenido pero que difieran en nombre, serán referenciados como un mismo archivo. En la Figura 2 se puede observar un ejemplo de este mecanismo.

En este paper, objeto, documento y archivo, serán utilizadas indistintamente. Además se utilizaran las siguientes definiciones:

- ▷ *CPB* es el ***Conjunto de Palabras a Buscar***.
- ▷ *DPF* es el ***Diccionario de Palabras Frecuentes***. Un diccionario creado a partir de las palabras populares. Se explica en detalle en la Sección 3.2 Balance de Carga.
- ▷ *Superllave* es la palabra elegida para iniciar la búsqueda. A partir de su hash se halla el nodo que administra la lista de objetos asociados a ella. La forma de elegirla se debe a la popularidad de las palabras, esto es, se utiliza como *Superllave* alguna de las contenidas en el *CPB* que no aparezcan el en *DPF*.

Si bien las ventajas de los índices invertidos son muy importantes, hay inconvenientes que deben ser solucionados para mejorar el sistema. A continuación presentaremos propuestas que solucionan dichos inconvenientes.

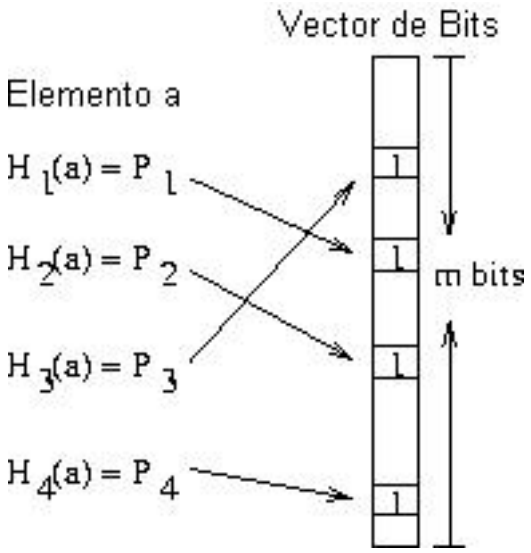

Figura 3: Un bloom filter donde se calculan cuatro funciones hash por palabra.

3.1. Sobrecarga

Este inconveniente surge debido a que en una consulta con k palabras se necesitan las respuestas de k nodos, donde cada respuesta contiene muchos documentos que luego serán descartados al hacer la intersección. El movimiento de tantos datos sobrecarga la red, y la operación de intersección consume demasiados recursos.

Para evitar la sobrecarga, MamUSHkA elimina la necesidad de consultar a k nodos, utilizando un sistema donde la consulta es enviada a solo uno de los nodos. El mecanismo se basa en obtener la key a partir del hash de la *Superllave*, y enviar el pedido con todas las palabras al nodo N encargado de dicha key. N recorre su lista de documentos en busca de resultados que contengan todas las palabras y los retorna.

Para mejorar la búsqueda que debe realizar N, se utilizan bloom filters calculados a partir de información del objeto, como nombre de archivo y/o etiquetas.

Un bloom filter es una estructura de datos eficiente en espacio que es usado para testear si un elemento es miembro de un conjunto. Los falsos positivos son posibles, pero no los falsos negativos. Mientras más elementos se agregan al bloom filter, más falsos positivos puede haber.

Un bloom filter vació es un arreglo de bits de m bits, todos seteados a 0. También debe haber k funciones hash diferentes, cada una de las cuales mapea un elemento a una de las m posiciones del arreglo.

Para agregar un elemento, se calculan los k hashes para obtener k posiciones en el arreglo, estas posiciones se setean a 1. Ver Figura 3.

Para consultar si un elemento está incluido, se calculan los k hashes del elemento para obtener k posiciones en el arreglo. Si alguno de los bits en estas posiciones es 0, entonces el elemento no está en el conjunto, caso contrario, el elemento podría estar. Que los bits en las k posiciones estén seteados a 1 no implica que el elemento pertenezca al conjunto, debido a que la inserción de otros elementos pudo haber seteado los bits en esas posiciones.

El nodo mantiene una tabla (Figura 4) indexada por la cantidad de unos u que tiene un bloom filter. Cada entrada de la tabla contiene la lista de bloom filters con u cantidad de unos, esta lista está ordenada numéricamente (tomando el bloom filter como un número) de mayor a

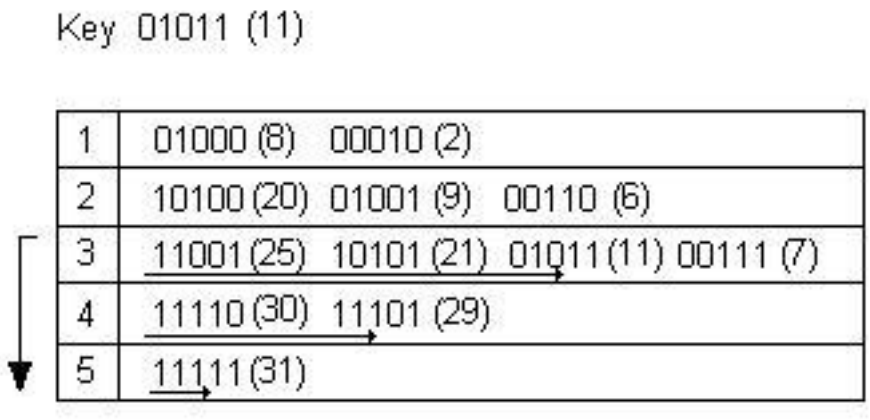

Figura 4: Una tabla de bloom filters donde se utilizan 5 bits para el arreglo. La frase a buscar generan el bloom filter 01011 que tiene tres unos y representa el número 11 en decimal. La búsqueda se inicia en la entrada 3 de la tabla (debido a los tres unos del bloom filter) y se examinan los valores 11001 (25), 10101 (21) y 01011 (11), de los cuales, solo se retornará el documento asociado a 01011 que coincide en todos sus unos. Como se observa, el valor 00111 (7) no se examina, debido a que 7 es menor que 11. La búsqueda continúa en las entradas 4 y 5 de la tabla, de las cuales, solo se retorna el archivo asociado a 11111 (31)

menor.

Cuando N recibe un pedido de inserción (el cual incluye el metadato, el bloom filter y la información de locación), este examina el bloom filter contando la cantidad de unos para obtener la entrada de la tabla, donde se encuentra la lista en la que debe ser insertado. La posición en la lista donde se insertará el bloom filter depende del valor numérico de éste.

Las consultas pueden ser simples o específicas. Las simples solo incluyen el *CPB*. Las específicas, además incluyen parámetros (como artista, álbum, etc) que restringen los resultados.

Cuando el pedido es de consulta, el nodo calcula el Bloom Filter *BFc* a partir del *CPB* y se cuenta la cantidad de unos del *BFc* para hallar la entrada de la tabla correspondiente. Se recorre la lista y se examinan los bloom filters *BFl* cuyos valores numéricos son mayores al de *BFc*. El examen consiste en hacer un *AND* entre *BFl* y *BFc*, y comparar este resultado con *BFc*. Si son iguales, quiere decir que *BFc* está incluido en *BFl*, por lo tanto es un resultado a devolver.De la misma forma se deben recorrer las listas de las entradas de la tabla cuya cantidad de unos sea mayor a la cantidad de unos del *BFc*.

En las búsquedas específicas, además de hacer un *AND* entre *BFl* y *BFc*, se examina que los parámetros adicionales coincidan con los parámetros de los metadatos asociados al bloom filter.

El fundamento detrás de este mecanismo es que un documento contiene las palabras buscadas si su bloom filter tiene seteado en uno al menos las mismas posiciones que el bloom filter de la consulta.

Debido a que la forma de indexar la tabla es por cantidad de unos, las listas de cada entrada tendrán valores cuyas posiciones de bits seteados a uno pueden diferir con el valor buscado, y dado que un bloom filter *BFc* está contenido en otro *BFl* si y solo si el valor numérico de *BFc* es menor al de *BFl* (porque si un filtro contiene a otro, quiere decir que posee unos en al menos las mismas posiciones o en más, lo que da como resultado un número más grande), solo

se examinan los *BFl* cuyo valor numérico sean mayores que el de *BFc*.

3.2. Balance de Carga

La frecuencia de las palabras (la cuenta de apariciones de palabras en objetos) varia enormemente. La distribución típicamente sigue la *ley de Zipf*, lo que significa que algunas palabras aparecen muy seguido mientras algunas otras, aparecen raramente. Esto genera hot spots, nodos que son consultados muy frecuentemente, y con grandes listas de objetos, para las cuales se debe destinar gran espacio de almacenamiento; además, procesar estas listas es más complejo, debido a que el esfuerzo es proporcional al tamaño de la lista.

Para aliviar la carga de consultas en los hot spots, se utiliza un ***Diccionario de Palabras Frecuentes*** (*DPF*). El *DPF* es una estructura de datos distribuida, en el cual, cada nodo puede registrar palabras frecuentes. Cuando un nodo descubre que la cantidad de documentos que tiene asignados una palabra supera un valor preestablecido u, este agrega la palabra al *DPF*. El contenido del diccionario es replicado y propagado entre los nodos de la red. Cada nodo mantiene un *DPF* local que periódicamente intercambia mensajes de actualización con los *DPF*s de los vecinos. De esta forma, después de un período de tiempo bien definido, el registro de una palabra k será propagado a todos los nodos de la red.

Aunque mantener este diccionario parezca costoso, no lo es realmente, ya que la *ley de Zipf* demuestra que solo una pequeña fracción de las palabras son muy comunes y deben ser insertadas en el *DPF*. Debido a esto, una vez que la red se estabilice, la inserción de palabras en el *DPF* será muy esporádica.

Para realizar una consulta, se elige una de las palabras del *CPB* que no se encuentre en el *DPF*. Esta palabra se denominará *Superllave*. La idea detrás de esto es evitar consultar a nodos que manejan palabras populares, que de otra forma serían sobrecargados con consultas. De esta forma, además, se consultan nodos con listas de objetos más pequeñas, lo que disminuye el tiempo de respuesta.

Mientras una palabra no sea declarada popular, esta es administrada por un solo nodo N, el cual acepta inserciones y consultas. Cuando el nodo advierte que una palabra P es popular (se supera un limite L de documentos), la ingresa en el *DPF*. A partir de aquí, las inserciones y consultas son tratadas de manera diferente.

Para que la lista de documentos en N no continúe creciendo hasta un máximo U (U > L, ver Figura 5), las inserciones son repartidas entre k nodos. Para esto, se consideran k hashes distintos de la misma palabra (entre ellos está el hash H que se utiliza normalmente). Esto es, cuando un nodo desea insertar un documento que contiene P, este elige al azar una función h_i de entre k funciones hash disponibles. Para la elección de este h_i se tendrá en cuenta que H tenga menos probabilidades de ser elegida. Lo que se busca es que se realicen la mayoría de las inserciones en nodos que no sean N, pero seguir insertando esporádicamente en ese nodo, debido a que su carga se irá aliviando a medida que los objetos que este mantiene vayan desapareciendo de la red. Si alguno de estos nodos ya contiene U documentos, entonces no aceptará más inserciones, por lo que el nodo que desea insertar, deberá elegir un nuevo hash para insertar el documento en alguno de los k-1 restantes.

Si se deben realizar consultas de palabras populares (algo que no sucederá a menudo por lo explicado anteriormente), se deberá consultar a los k nodos involucrados para obtener todos los resultados. Con esto, además de no saturar a un nodo con una lista enorme, se logra que las consultas sean procesadas más rápido gracias a que serán llevadas a cabo en paralelo por los k nodos.

Figura 5: Aquí se observan los límites L y U. Cuando un nodo se encuentra en la zona 1, este acepta inserción de metadatos; cuando el límite L es superado y se entra en la zona 2, cada vez que el nodo recibe un pedido de inserción, este acepta e inserta el metadato, pero avisa al nodo que inició el pedido de que se encuentra en la zona 2. Cuando el nodo supera el límite U y entra en la zona 3, este deja de aceptar inserciones.

3.3. Solo búsquedas exactas

La estructura de índices invertidos usando hash de palabras no permite buscar palabras mal escritas o con errores ortográficos. Este problema se debe a que las búsquedas no se realizan por palabras sino por hash de palabras, y por definición, dos hash aplicados a palabras similares, darán resultados totalmente distintos.

Definimos palabra inexacta como aquella que no expresa lo que el usuario desea, ya sea por errores ortográficos, de tipeo o por desconocimiento (por ejemplo cuando no se conoce como se escribe el nombre de un artista).

Para la solución propuesta se utiliza un ***Diccionario de Palabras Locales*** (*DPL*), que se va construyendo con la información de los objetos insertados. Esto es, al llegar un pedido de inserción al nodo, se extraen el nombre y las etiquetas del objeto, y se insertan aquellas palabras que aún no pertenezcan al *DPL*.

El mecanismo utilizado para permitir búsquedas con palabras inexactas es iniciar la búsqueda de forma normal, es decir, hallar el nodo encargado de una *Superllave*. Cuando la consulta llega al nodo, éste verifica que las palabras del *CPB* pertenezcan al *DPL*, las que no están incluidas se intentan corregir y se arma un nuevo *CPB*. Luego de este chequeo, la consulta continuará como se describió en secciones anteriores, pero utilizando el nuevo *CPB*.

La corrección de palabras se logra utilizando un algoritmo basado en la *distancia Levenshtein*, también conocida como *edit distance*, que es una métrica para medir la cantidad de diferencias entre dos palabras. Esta diferencia está dada por el mínimo número de operaciones necesarias para transformar una palabra en otra, donde las operaciones son inserción, borrado o sustitución de un carácter, y transposición de dos caracteres.

El algoritmo genera el conjunto de palabras que se encuentran a una distancia d de la palabra a corregir, de este conjunto descarta las que no pertenezcan al *DPL* y entre las restantes selecciona la de mayor frecuencia. Con una alta probabilidad la palabra elegida será la que se deseaba. Este algoritmo es la base de los correctores ortográficos utilizados por Google, Office, Mozilla, etc.

De esta forma, si se cometieron errores ortográficos o no se conocía el nombre con exactitud, igualmente, con una gran probabilidad, se retornaran los resultados que se deseaban encontrar.

En caso de que la *Superllave* sea inexacta, el nodo NC asociado a esta, muy probablemente, no contendrá objetos con dicha *Superllave*, a no ser que las palabras del objeto deseado también estén mal escritas. En esta situación, NC retornará un mensaje "*posible inexactitud*". Al recibir este mensaje, se realiza nuevamente la búsqueda eligiendo una nueva *Superllave* de entre las palabras restantes del *CPB*. Este proceso continúa hasta que el mensaje recibido sea distinto a "*posible inexactitud*", se hayan utilizado todas las palabras del *CPB* como *Superllave*, u ocurra un error.

Con este mecanismo, si al menos una de las palabras del *CPB* es correcta, probablemente se obtendrán resultados deseados.

3.4. Unico Punto de Falla

Un objeto que contiene las palabras w_1, w_2, ..., w_n, crea entradas por cada palabra, lo que significa que la información acerca del objeto está repetidamente guardada en n lugares diferentes.

A pesar de que un objeto es indexado desde múltiples lugares, cada palabra sigue siendo manejada por un solo nodo. Cualquier falla del nodo negará todas las consultas que involucran esa palabra.

MamUSHkA esta pensado para implementarse sobre una red subyacente que provea redundancia, como por ejemplo Kademlia, donde los valores son almacenados en varios nodos (k de ellos) para permitir que los nodos entren y salgan y aún así tener los valores disponibles en algún nodo. Cada cierto tiempo, un nodo que almacena un valor explorará la red para encontrar los k nodos más cercanos a un hash dado y replicará el valor en ellos.

4. Conclusión y Trabajos Futuros

Hemos presentado un sistema de búsqueda basado en una estructura de índices invertidos, aprovechando sus ventajas y agregando mecanismos para solucionar sus problemas. Con estas mejoras se logra aliviar la carga de los nodos que manejan palabras populares (Hot Spots), disminuir la carga en la red, acelerar la búsqueda de resultados dentro del nodo y agregar inteligencia a esta.

Con el *DPF* se trata de evitar consultar a nodos NP que administran palabras populares, no tomando como *Superllave* a una palabra que pertenezca al *DPF*. De esta forma, las búsquedas estarán más repartidas. Además, distribuyendo las listas de objetos de los NPs entre varios nodos, se evita sobrecargarlos con inserciones y acelerar las búsquedas en estos, ya que se realizaran en paralelo y las listas a recorrer serán más cortas.

Con el formato de consulta a un solo nodo se disminuye la carga en la red y se elimina la necesidad de intersectar las listas de objetos de los nodos que manejan cada palabra del *CPB*.

Utilizando Bloom Filters, se disminuye el trabajo que debe realizar un nodo para encontrar resultados correspondientes a las consultas, por lo tanto se disminuye además el tiempo de respuesta. El mecanismo utilizado permite también devolver objetos que contengan más palabras que el *CPB*.

Gracias al *DPL* y al algoritmo de corrección de errores, se añade inteligencia a la búsqueda, intentando adivinar lo que el usuario quiso buscar, en el caso de que alguna/s palabra/s de la consulta no se encuentre en ningún objeto.

El proyecto se está comenzando a implementar en el simulador PlanetSim [14, 15], el cual fue elegido después de considerar diversos simuladores, entre ellos PeerSim [16], OverSim [17] y GPS [18]. Las características más destacadas de PlanetSim son su muy buena división en módulos, lo cual facilita el entendimiento y la implementación de aplicaciones; el proveer una fácil transición del código de simulación al código de una implementación real, y poseer una buena documentación.

Ya se cuenta con una implementación de bloom filters, y del corrector ortográfico, el cual es un código en java [19] adaptado del código en python original provisto por [20] La base de datos que se utilizará para realizar las pruebas es Netflix [21], una base de datos de películas con 17.770 títulos de películas.

Referencias

[1] Ion Stoica, Robert Morris, David R. Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, pages 149–160, 2001.

[2] *Chord (distributed hash table)*. http://en.wikipedia.org/wiki/Chord_%28DHT%29.

[3] Maymounkov and Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. In *International Workshop on Peer-to-Peer Systems (IPTPS), LNCS*, volume 1, 2002.

[4] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content addressable network. Technical Report TR-00-010, International Computer Science Institute, Berkeley, CA, Oct 2000.

[5] Patrick Reynolds and Amin Vahdat. Efficient peer-to-peer keyword searching. In Markus Endler and Douglas C. Schmidt, editors, *Middleware*, volume 2672 of *Lecture Notes in Computer Science*, pages 21–40. Springer, 2003.

[6] 1977 Omprakash D. (Omprakash Dev) Gnawali. A keyword-set search system for peer-to-peer networks. Master's thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 2002.

[7] R. Brunner. A performance evaluation of the kad-protocol. Master's thesis, Mannheim University, Germany, 2006.

[8] Thomas Bocek, Ela Hunt, David Hausheer, and Burkhard Stiller. Fast similarity search in peer-to-peer networks. Technical report, IEEE/IFIP Network Operations and Management Symposium, Salvador da Bahia, Brazil, April 2008.

[9] L. Liu and K. D. Ryu. Supporting efficient keyword-based file search in peer-to-peer file sharing systems. Technical report, IBM Research, 2004.

[10] Ashish Gupta, Manan Sanghi, Peter Dinda, and Fabian Bustamante. Magnolia: A novel dht architecture for keyword-based searching, May 2005.

[11] Shi, Yang, Wang, Yu, Qu, and Chen. Making peer-to-peer keyword searching feasible using multi-level partitioning. In *International Workshop on Peer-to-Peer Systems (IPTPS), LNCS*, volume 3, 2004.

[12] Hai Zhuge and Liang Feng. Distributed suffix tree overlay for peer-to-peer search. *IEEE Trans. Knowl. Data Eng*, 20(2):276–285, 2008.

[13] Joung, Yang, and Fang. Keyword search in DHT based peer-to-peer networks. *IEEEJSAC: IEEE Journal on Selected Areas in Communications*, 25, 2007.

[14] Pedro García López, Carles Pairot, Rubén Mondéjar, Jordi Pujol Ahulló, Helio Tejedor, and Robert Rallo. Planetsim: A new overlay network simulation framework. In Thomas Gschwind and Cecilia Mascolo, editors, *SEM*, volume 3437 of *Lecture Notes in Computer Science*, pages 123–136. Springer, 2004.

[15] Jordi Pujol Ahulló, Pedro García López, Marc Sànchez Artigas, Marcel Arrufat Arias, Gerard París Aixalà, and Max Bruchmann. Planetsim: An extensible framework for overlay network and services simulations. Technical report, 2007.

[16] Peersim. http://peersim.sourceforge.net.

[17] Ingmar Baumgart, Bernhard Heep, and Stephan Krause. Oversim: A flexible overlay network simulation framework. In *Proceedings of 10th IEEE Global Internet Symposium (GI '07) in conjunction with IEEE INFOCOM 2007, Anchorage, AK, USA*, pages 79–84, May 2007.

[18] Weishuai Yang and Nael B. Abu-Ghazaleh. GPS: A general peer-to-peer simulator and its use for modeling bittorrent. In *MASCOTS*, pages 425–434. IEEE Computer Society, 2005.

[19] Spelling corrector (java). http://www.raelcunha.com/spell-correct.php.

[20] Spelling corrector (python). http://www.norvig.com/spell-correct.html.

[21] Netflix. http://www.netflixprize.com/.